# Programming With Infrared Sockets


## California Software Laboratories

**Your Outsourcing Partner**

**California Software Labs**
**96 Corporate Park,**
**Suite 100,**
**Irvine, CA 92606,**
**USA.**

**Phone**
**(949) 474-1363**
**Fax**
**(949) 474-1362**

www.cswl.com
info@cswl.com

# Programming With Infrared Sockets

**A Technical Report**

**Technical Expertise Level :** Intermediate
**Requires knowledge of :** Windows Sockets

Written by

*Prasanna .V, Software Engineer.*

***December 21, 1998***

***INDEX***

## Introduction

The term IrDA (Infra Red Data Association) refers to the international body of about 150 countries which aims at setting standards for device communications that use Infrared beams - to give the users access to high quality Infrared communications. They conduct necessary tests and issue guidelines for various hardware and software that are designed to be IrDA-compliant. Developers and solution providers for Infrared communications should get the "IrDA-compliant Logo" to claim that their products are IrDA-compliant. Infrared communication is mainly targeted towards HandHelds, PalmPCs and other devices that have a built-in Infrared port. Printers with IR ports enable us to print without any external power.

## What is IRDA ?

Enough has already been talked about the TCP /IP communications and windows sockets. IrDA, and Infrared communication standard, being wireless and mobile oriented, calls for a different model where there are no IP addresses or DNS servers, where the network is dynamic and devices enter and leave the network frequently. The conventional machine name-to-IP address mapping is absent and instead we have a service name /LSAP-SEL combination. Every IrDA device is identified by its name and an unique ID. Software (applications) within IRDA devices connect to applications on other devices by using a service name or an LSAP - SEL number registered by a target application on that device. Whenever such a service is created, an entry is made in a database called IAS, containing the service name along with the next available LSAP-SEL number (discussed later). This database can be queried by other devices to find the LSAP-SEL and establish communication.

The devices of interest include embedded devices, hand-helds, laptops etc, which have at least one built-in IR port. These IrDA-enabled devices can be used to build security systems, connect to a LAN and obtain the various services available on the net etc., Printers with IR ports can be installed and used with ease (walk-up printing). IrDA also enables us to connect our Mobile phone and our H/PC. We can send and receive faxes, E-mails etc., by establishing IR communications between the latest GSM (Global Systems and Mobile Communications) Mobile phone and the HandHeld running Windows CE. The phone in this case acts like a Modem and the IrDA protocol is used for communication purposes. For more details on GSM phones & IR, move to www.windows.hess.net

## Features of IRDA

The following are some features of IrDA worth mentioning.

- Easy to implement and simple to use
- Safe in any environment
- No electromagnetic noise
- No government regulatory issues
- Minimum crosstalk

Support for IrDA is available in Windows 98 and is expected in Windows NT 5.0. Although Infrared device drivers are available for Windows 95, the support is not complete as I could not create an Infra red socket even after installing the Infra red driver. This is because the windows sockets DLLs that comes along with Windows 95 doesn't provide any support for IRSock.

Manufacturers like ActiSYS Corp (www.actisys.com) provide IR adapters and add-on cards for IR communications. These are IrDA compliant. The adapters (called dongles) should be connected to the RS-232 serial port. A typical such adapter

enables reliable IR links for distances up to 2.4 meters. In order for the IR devices to communicate with each other, the Infrared window of the devices should be facing each other. The line of sight should be free of obstructions like excessive light in the operating area, vibrations etc., The beam of infrared is largely affected by such obstructions which may gradually weaken the communication and finally end up disconnecting the devices in action.

The good thing is that we in Calsoft/CSWL are reasonably aware of all the supporting technologies.

IrDA communication is bi-directional but half-duplex. IdDA supports three standards, Slow IR (SIR), Medium IR (MIR) and Fast IR(FIR). The Slow IR standard specifies a maximum speed of 115.2 KBPS and the maximum size of a data packet is 2KB. The Fast IR specifies a maximum speed of up to 4 Mbps. The processing requirements of FIR call for a dedicated add-on card on your PC.

Currently, under the Windows implementation of the IrDA stack, we find there can be only one line of communication between two IR devices. If the source or the destination device has to be changed, then the connection has to be dropped and a fresh connection made.

## IRDA Implementation

From a programmer's perspective, IrDA on Windows has three things to offer.

The IRSock, an extension provided by Windows sockets.

The IRCOMM, a protocol that enables applications use the conventional communication APIs to perform IR communications without knowing much about IRSock or IRDA. The IRCOMM abstracts the underlying techniques used to enable

IR communications and lets the applications address the serial or parallel ports as usual and does the required groundwork to map these calls suitably to enable IR communications.

The IrLAN, a protocol which helps to connect our embedded systems to Local Area Networks.

The focus of this article is on IRSock and IrCOMM and any discussion on IrLAN is beyond the scope of this article.

**The IRSOCK**

The IRSock is actually an extension provided by the Windows Sockets specification version 1.1. The IRSock is meant for those devices that communicate through IRDA. The term "IRSock" may sound odd for any one who hears the term for the first time. It is odd indeed. Though basically not different from the windows sockets in many ways, it differs from windows sockets primarily in two issues.

Name Resolution
Method of addressing

The network of IRDA devices is dynamic i.e., we don't have a fixed number of devices that are always tied to the network. Rather, devices frequently move in and out of the network. In such a situation, we can't have fixed networks with Domain servers, which could map the machine name to its corresponding IP address. So how does a device identify another?

The answer goes as follows.

An IR device usually looks for any IR devices in its vacinity. If found, it connects to that device by specifying its name or LSAP-SEL(not recommended). (For details on LSAP - SELs, see Appendix). Various options are need to be set so that one IR device can discover the IR device in its vicinity and get connected to it. This being the various basic thing behind the IR connectivity, let's go further to discuss the IRDA protocol stack and how the applications use the protocol stack.

Another important feature of IRSock, which distinguishes it from Winsock, is that IRSock doesn't support Datagram sockets. The reliable stream socket is the only option available.

## Windows CE and IRDA

Windows CE Version 1.0 is the first operating system to support IrDA. It is well built to support a wide range of communication options, including IrDA. Windows CE supports the following modes of communication through IR.

Windows CE supports "raw IR ", where the IR adapter is connected to a COM port such as COM1 or COM2 and these ports can be accessed as usual using our serial communication APIs. The raw IR mode lets the programmer control the entire the sequence of the data flow. No handshaking or discovery happens here as we have in our usual IR communications. No collision detection mechanisms are available and so extra care should be taken when working in this mode.

Another mode of interest is the IRCOMM mode, where we can enable our applications which use to address the COM ports use the services of IRDA. The IRCOMM performs the necessary internal mappings for this. The application will be using the usual serial communication APIs but the IRDA stack will be put to work in the background. See the section "IRCOMM" for more details.

WindowsCE supports IRSock, which is a more efficient mode of IR communications. For details on IRSock, refer to "The IRSOCK " section.

## IRSOCK

Having discussed briefly about the underlying protocol stack, it's time now to get onto the various programming aspects of IRSock.

The Microsoft Foundation Classes (MFC) doesn't provide any support for IRSocks as it does for Windows Sockets. So we have to use the various Win32 API functions for our purpose. However, developers are free to develop their own classes, which anyway are going to use the APIs only.

### Basic Requirements

Users have two options for IRSock programming.

***1. HandHeld -To - HandHeld Communication***

If the users need to have both their client and server applications in HandHelds running Windows CE, they require a PC running Windows NT version 4.0 or later along with VC ++ 5.0 and VC++ Toolkit for Windows CE to develop their IRSock applications. To download their IRSock applications to the HandHelds, they should install Windows CE Services.

After downloading their applications to the HandHelds, the HandHelds, which have a built-in IR port, can be made communicate with each other through IRSock.

***2. HandHeld - DeskTop PC communication***

If the users like to have their client or server applications running on the DeskTop and the other one in a HandHeld, they should install Windows 98 which provides complete support for IRSock. Users should add the Infrared Device as given in the ACTiSYS Reference Manual www.actisys.com. To develop the application that runs in the HandHeld, the requirements are the same as mentioned in the "HandHeld-To-HandHeld Communication" section. After the application has been moved to the HandHeld, the DeskTop PC and the HandHeld are free to communicate.

**Coding Aspects**

To start with, lets see how the programming for IRSockets differ from that for Windows Sockets.

We usually build our sockets through the socket() call with a AF_INET address family, in case, if we are using APIs in our applications. The socket may be a stream socket or a datagram socket. IRSock differs from windows sockets in both the above mentioned aspects. It requires a special address family named AF_IRDA. Moreover, an IRSock can only be a stream socket and not a datagram socket. The IRSock requires a header file named AF_IRDA.h for its operations. This header file is available with the VC++ toolkit for Windows CE and IRD3DDk. Windows NT Version 5.0 Beta 1 provides a uniform header file which provides support for the three platforms namely, Windows NT, Windows 98 and WindowsCE.(See appendix for moving to the header file).

_WIN32_WINNT
_WIN32_WCE
_WIN32_WINDOWS

After creating the socket, we bind our socket to a particular address using the bind() call and the SOCKADDR_IN address structure, in case of a server application. For IRDA, the structure is SOCKADDR_IRDA, which has the following structure.

*Struct SOCKADDR_IRDA {*
*u_short IRDAAddressFamily;*
*u_char IRDADeviceID[4];*
*char IRDAServiceName[25];*
*};*

The IRDAAddressFamily should be set to AF_IRDA. The IRDADeviceID is an array used to hold an unique ID for the IR device. For an IRSock server, the ID is usually ignored. The third member, irdaServiceName shall be an user-defined string or a predefined string of format "LSAP-SELxxx" where xxx can be a number in the range 0-127. Since the allowable range is small, it is advisable to use an user-defined string for the device identification.

After the bind() call, an entry will be added to the IAS (Information Access Service) with the name specified by the user-defined string and an attribute value equal to the next available LSAP-SEL . Place the server in the listen mode using listen() call as usual and the rest of the things are no different from what we have in windows sockets.

This being the case of the server, we now see a how an IR client is built.

**The IRSock Client**

The way by which a machine running an IRSock client gets connected to the server is altogether a different one. After the socket is created, the client is made to search

for the IR devices within the permissible range. This is done by the getsockopt() call. The API has the following structure.

*int getsockopt(SOCKET s,int level,int*
*optname,const char FAR *optval,int optlen)*

The parameters of interest here are the level and the optname. To enable the device search for other devices, the level should be set to SOL_IRLMP and the optname parameter can be one of the following.

IRLMP_ENUMDEVICES, which enumerates remote IRDA devices
IRLMP_IAS_QUERY ,which queries IAS attributes
IRLMP_SEND_PDU_LEN, which queries the maximum size of send packet for IrLPT mode

If the parameter IRLMP_ENUMDEVICES is used, the result is the information about all the devices available in the range in the form of a data structure named DEVICELIST whose structure is given as follows.

*struct DEVICELIST*
*{*
*ULONG numDevice;*
*IRDA_DEVICE_INFO Device[1];*
*};*

*The IRDA_DEVICE_INFO structure is given as follows.*

*struct IRDA_DEVICE_INFO*
*{*
*u_char IrdaDeviceID[4];*
*char irdaDeviceName[22];*

*u_char Reserved[22];*
*};*

The client shall then issue a connect() using the information returned above. When the client attempts to connect, an IAS lookup is performed using the GetValueByClass () service to get the LSAP-SEL for the service name specified in the SOCKADDR_IRDA structure. Next the client is connected to that service point. If the LSAP-SEL number is directly specified instead of the service name in the SOCKADDR_IRDA structure, the IAS lookup is bypassed and the client is directly connected to that service point.

The above method of using getsockopt() to look for devices in the range results in busy-waiting and will consume battery power because the function call has to be put in a loop to keep looking for the devices in range. An alternative way of achieving the same is through the use of setsockopt() call. By this call, we can set our IR device in discovery mode(a mode of operation where the IR device looks for other IR devices in its vicinity) using the IRLMP_DISCOVERY_MODE option and make the client trace the other IR device in action. (See appendix for the program segment )

Note: IRLMP_DISCOVERY_MODE does not seem to be supported by WindowsCE as of today. The af_irda.h header file also seems to indicate this. Therefore explicit calls to Sleep() within the discovery loop will have to be used to reduce the CPU utilization.

Another value of importance the optname parameter of the getsockopt() can hold is IRLMP_IAS_QUERY. By specifying this value, the client can query for various services available on the server. It can retrieve attributes like the LSAP-SEL number for a particular service, the service name and so on.

Other concepts like the blocking and non-blocking sockets are the same even if the socket is an IR Socket.

## IRCOMM

The IrCOMM layer is at the top of the IrDA protocol stack and provides an emulation of a device connected via a serial or parallel port. Legacy applications proceed to work the same way in communicating with the devices through the same APIs without knowing that it is actually the IrDA protocol stack that is put to operation. By this way, older applications are made to make use of the latest and efficient means of communications available.

The IrCOMM emulates the RS-232 serial ports and the Centronics parallel interface port. When an application writes to a COM port, IrCOMM stands midway and uses the services of the IrDA protocol stack to perform the required communication with the IR device. The state changes occurring at the serial port are converted into messages suitable to the underlying protocol and transmitted to the other device connected to it through IR. The IrDA protocols provide full flow control between the two devices in action.

**IrCOMM modes**

IrCOMM operates in the following modes

- 9 wire mode
  In this mode, the IrCOMM supports the transport of state change messages.

- IrLPT mode
  This mode is used to communicate with printers through IR.

- 3 Wire mode
  This mode, together with the IrLPT mode makes the IrDA protocol run in exclusive mode, thereby making them accessible by only one application at a time.

- 3 Wire cooked mode
  This mode neither prevents other applications from accessing the IrDA stack nor does it transport status messages.

**IrCOMM Programming**

See appendix for the program segment, which explains, on how to implement an IrCOMM server and client.

## Conclusion

To know more about the sample hardware required for IR communications, visit www.actisys.com. More over, any discussion on specifications for the layers of the IRDA protocol stack is beyond the scope of this article. For those details, visit the ftp.irda.org where complete documentation of all the protocol layers can be found.

## Appendix

**Blocking and Non Blocking Sockets**

A socket, be it a windows socket or an IR Socket, by default, works in blocking mode. That is, a function call like recv() won't return until all the data has been read. This places some restrictions as the application won't be able to continue with its execution till the function returns. A solution for this problem can be made possible through the use of asynchronous or non-blocking sockets. These sockets return immediately irrespective of whether the operation is completed or not.

To set a socket in non-blocking mode, we use the API ioctlsocket() which has the following structure.

int ioctlsocket (SOCKET sock, long command, u_long *arg)

The command parameter can take any one of the following parameters.

FIONBIO - Sets or clears the socket's blocking mode
FIONREAD - Returns the number of bytes that can be retrieved by the recv () function.

By setting the command parameter to FIONBIO and making the arg parameter point to a non-zero value, we can set the socket in non-blocking mode.

The restriction the sockets place when operating in the non-blocking mode is that they have to be polled continuously to check for the arrival of incoming connections, data etc., which has a severe impact on CPU utilization as well as the longevity of the battery power. So it will better if the code that performs the process of receiving data is implemented as a separate thread. Otherwise the use of WSAAsyncSelect() API call is recommended (this is not currently supported by CE). The call enables the user to specify the network events for which the application can receive notification.

**Logical Service Access Point Selectors (LSAP-SELs)**

LSAP-SELs are integers that range between 0-127. Users, when programming for IR Sockets, shall specify the either the name of the service or the LSAP-SEL number to connect to the server. Whenever the user attempts to connect to a server by specifying its service name in the SOCKADDR_IRDA structure, the IAS maps

the service name to its corresponding LSAP-SEL and the connection is established to that LSAP-SEL. Direct usage of the LSAP-SEL is discouraged because of the small range available. The following line of code shows how to specify an LSAP-SEL number directly in the SOCKADDR_IRDA structure. SOCKADDR_IRDA address = {AF_IRDA, 0,0,0,0, "LSAP-SEL30"};

**A Sample IR Client that works in the Discovery Mode**

```
Main ()
{
SOCKET sock;
SOCKADDR_IRDA address = {AF_IRDA, 0, 0,0,0,"MyServer"};
DEVICELIST devList;
char helloClient[25];
int mode = TRUE;

sock = socket (AF_IRDA, SOCK_STREAM, 0);
// place socket in discovery mode
setsockopt (sock, SOL_IRLMP, IRLMP_DISCOVERY_MODE, (char
*)&devList,&mode);
// wait for a new device to come into range; //blocking call.
recv(sock, (char *)&devList, sizeof(devList), 0);
// turns discovery mode off
mode = FALSE;
setsockopt (sock, SOL_IRLMP, IRLMP_DISCOVERY_MODE, (char *) &devList,
&mode);
// choose the first device in the list
for(int i=0;i<=3;i
Address.irdaDeviceID [i] = devList. Device [i]. irdaDeviceID;
// connect this client to the server on the //device with service name MyServer
```

```
connect(socket, (struct SOCKADDR *)&address, sizeof(SOCKADDR_IRDA));
// send it a greeting
send(sock, "Hello Server!", strlen("Hello Server!")+1, 0);
// wait for a response; blocking call.
recv(sock, helloClient, sizeof(helloClient), 0);
printf("%s\n", helloClient);
closesocket(sock);
}
```

Back to Index

*A Sample IrCOMM Server*

*The following code snippet builds a simple IrCOMM Server*

```
#define IAS_SET_ATTRIB_MAX_LEN 32

// buffer for IAS set
BYTE IASSetBuff[sizeof (IAS_SET) - 3 + IAS_SET_ATTRIB_MAX_LEN];
int IASSetLen = sizeof (IASSetBuff);
PIAS_SET pIASSet = (PIAS_SET) &IASSetBuff;

// for the setsockopt call to enable 9 wire IrCOMM
int Enable9WireMode = 1;

// server sockaddr with IrCOMM name
SOCKADDR_IRDA ServSockAddr = { AF_IRDA, 0, 0, 0, 0,
"IrDA:IrCOMM" };
SOCKADDR_IRDA PeerSockAddr;
int sizeofSockAddr;
```

```
SOCKET ServSock;
SOCKET NewSock;

if ((ServSock = socket(AF_IRDA, SOCK_STREAM, 0)) == INVALID_SOCKET)
{
// WSAGetLastError
}

// add IrCOMM IAS attributes for 3 wire cooked and 9 wire raw, see IrCOMM spec
memcpy(&pIASSet->irdaClassName[0], "IrDA:IrCOMM", 12);
memcpy(&pIASSet->irdaAttribName[0], "Parameters", 11);

pIASSet->irdaAttribType = IAS_ATTRIB_OCTETSEQ;
pIASSet->irdaAttribute.irdaAttribOctetSeq.Len = 6;

memcpy(&pIASSet->irdaAttribute.irdaAttribOctetSeq.OctetSeq[0],
"\000\001\006\001\001\001", 6);

if (setsockopt(ServSock, SOL_IRLMP, IRLMP_IAS_SET, (const char *) pIASSet,
IASSetLen) == SOCKET_ERROR)
{
// WSAGetLastError
}

// enable 9wire mode before bind()
if (setsockopt(ServSock, SOL_IRLMP, IRLMP_9WIRE_MODE, (const char *)
&Enable9WireMode, sizeof(int)) == SOCKET_ERROR)
{
// WSAGetLastError
}
```

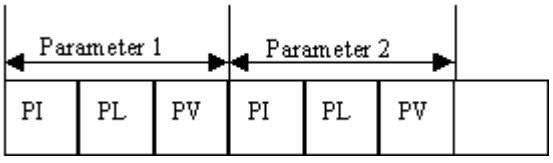
```
if (bind(ServSock, (const struct sockaddr *) &ServSockAddr,
sizeof(SOCKADDR_IRDA)) == SOCKET_ERROR)
{
// WSAGetLastError
}

// nothing special for IrCOMM from now on...
if (listen(ServSock, SERV_BACKLOG) == SOCKET_ERROR)
{
// WSAGetLastError
}
```

A brief discussion on the above snippet goes as follows.

The IrCOMM services have their class name in the IAS database as IrDA: IrCOMM. This should be specified as the class name when an IrCOMM service is to be created.

The "Parameters" is an attribute of "IrDA:IrCOMM" that is used to differentiate the various IrCOMM services running in a device. The attribute type is specified as IAS_ATTRIB_OCTET_SEQ, which is the type of the information held by the Parameters collection. The attribute value is an Octet sequence, which holds information about each and every parameter. The Parameters collection has the following format where each parameter is stored as a combination of three elements namely, The Parameter ID, The Parameter Length and The Parameter Value.



The Parameter Identifier and the Parameter Length are of size equal to 1 byte and the Parameter value has the size equal to the value held by the Parameter Length.

In our example, the first three octets 00,01,06 specify that the Parameter Identifier is 0x00 which refers to the ServiceType parameter (Used to specify the various services supported (3-Wire, 9-Wire etc.,), 01 is the Parameter Length which is one byte and 06 is the Parameter value, which is actually a bitmask to specify one of the four services.

Bit 0 - 3 Wire raw
Bit 1 - 3 Wire
Bit 2 - 9 Wire
Bit 3 - Centronics

Since the server in our example uses 3 - Wire and 9 - Wire services, the bits 1 and 2 are set giving a value of 6.

The next three octets convey the following meaning.
Parameter ID - 01 - ServicePortType parameter
Parameter Length - 01- one byte

The parameter value, if the parameter ID is 01, is a bitmask of 2 bits, with bit 0 for the Serial port and bit 1 for the parallel port.

The IAS_SET data structure is filled with the above elements and the setsockopt () call is used to set these options. The IRLMP_9WIRE_MODE is used to set the connection to the 9-wire mode. After this, the proceedings are exactly the same as that of the IrSock.

For more details on the possible values the elements of a parameter can hold, refer to the article," IRCOMM: Serial And Parallel Port Emulation over IR" available at ftp.irda.org.

**A Sample IrCOMM Client**

This snippet shows a client that connects in the 9 Wire mode.

```
#define DEVICE_LIST_LEN 5

// discovery buffer
BYTE DevListBuff[sizeof (DEVICELIST) - sizeof(IRDA_DEVICE_INFO) +
(sizeof (IRDA_DEVICE_INFO) * DEVICE_LIST_LEN)];
int DevListLen = sizeof(DevListBuff);
PDEVICELIST pDevList = (PDEVICELIST) &DevListBuff;
Int DevNum;

#define IAS_QUERY_ATTRIB_MAX_LEN 32

// buffer for IAS query
BYTE IASQueryBuff[sizeof(IAS_QUERY) - 3 +
IAS_QUERY_ATTRIB_MAX_LEN];
Int IASQueryLen = sizeof(IASQueryBuff);
PIAS_QUERY pIASQuery = (PIAS_QUERY) &IASQueryBuff;

// for searching through peers IAS response
BOOL Found = FALSE;
UCHAR *pPI, *pPL, *pPV;
```

```
// for the setsockopt call to enable 9 wire IrCOMM
int Enable9WireMode = 1;

SOCKADDR_IRDA DstAddrIR = { AF_IRDA, 0, 0, 0, 0, "IrDA:IrCOMM" };

if ((pConn->Sock = socket(AF_IRDA, SOCK_STREAM, 0)) ==
INVALID_SOCKET)
{
// WSAGetLastError
}

// search for the peer device
pDevList->numDevice = 0;
if (getsockopt(pConn->Sock, SOL_IRLMP, IRLMP_ENUMDEVICES, (CHAR *)
pDevList, &DevListLen)
== SOCKET_ERROR)
{
// WSAGetLastError
}

// if (pDevList->numDevice == 0)
{
// no devices found, tell the user
}

// assume first device, we should have a common dialog here
memcpy(&DstAddrIR.irdaDeviceID[0], &pDevList->Device[0].irdaDeviceID[0],
4);

// query the peer to check for 9wire IrCOMM support
```

```
memcpy(&pIASQuery->irdaDeviceID[0], &pDevList-
>Device[0].irdaDeviceID[0], 4);

// IrCOMM IAS attributes
memcpy(&pIASQuery->irdaClassName[0], "IrDA:IrCOMM", 12);
memcpy(&pIASQuery->irdaAttribName[0], "Parameters", 11);

if (getsockopt(pConn->Sock, SOL_IRLMP, IRLMP_IAS_QUERY, (char *)
pIASQuery,
&IASQueryLen) == SOCKET_ERROR)
{
// WSAGetLastError
}

if (pIASQuery->irdaAttribType != IAS_ATTRIB_OCTETSEQ)
{
// peer's IAS database entry for IrCOMM is bad
// error
}

if (pIASQuery->irdaAttribute.irdaAttribOctetSeq.Len <3)
{
// peer's IAS database entry for IrCOMM is bad
// error
}

// search for the PI value 0x00 and check 9 wire, see IrCOMM spec.
pPI = pIASQuery->irdaAttribute.irdaAttribOctetSeq.OctetSeq;
pPL = pPI + 1;
pPV = pPI + 2;
```

```
while (1)
{
if (*pPI == 0 && (*pPV & 0x04))
{
Found = TRUE;
break;
}

if (pPL + *pPL >= pIASQuery-irdaAttribute.irdaAttribOctetSeq.OctetSeq +
pIASQuery->irdaAttribute.irdaAttribOctetSeq.Len)
{
break;
}

pPI = pPL + *pPL;
pPL = pPI + 1;
pPV = pPI + 2;
}

if (! Found)
{
// peer doesn't support 9-wire mode
// error
}

// enable 9wire mode before connect()
if (setsockopt(ServSock, SOL_IRLMP, IRLMP_9WIRE_MODE, (const char *)
&Enable9WireMode, sizeof(int)) == SOCKET_ERROR)
{
// WSAGetLastError
}
```

```
// nothing special for IrCOMM from now on...
if (connect(pConn->Sock, (const struct sockaddr *) &DstAddrIR,
sizeof(SOCKADDR_IRDA))
== SOCKET_ERROR)
{
// WSAGetLastError
}
```

At the client end, a query is made for the availability of the IrCOMM services at the server. This is done by the call to the getsockopt () with IRLMP_IAS_QUERY option. The IAS_QUERY structure's irdaClassName member variable is assigned the string "IrDA:IrCOMM" and irdaAttribName is specified as "Parameters".

The Parameter Identifier should hold a value of 0x00 and the Parameter value should be 0x04 to indicate a service in 9 Wire mode. This is checked for to see whether the required service exists. If found, the setsockopt() is used to enable the 9 wire mode before a connect() call is issued. Then, things are the same as usual with the client calling connect() to connect to the server and proceeding to send and receive data.

**The AF_IRDA.h Header File**

Given here is the Af_irda.h header file, which supports all the three platforms namely, Windows NT 5.0, Windows 98 and Windows CE. To view the header file, click here.

## Bibliography

An introduction to the IRDA standard and system implementation-Feature article on IRDA , May 1996.
http://www.actisys.com//article/wsd.html

Windows CE offers exciting Mobile computing capabilities.
http://www.cetj.com/archives/9806/9806wind.shtml

Arne's Windows CE Page.
http://www.windowsce.hess.net/

Chris De Ferreira's Windows CE Site.
http://www.windowsce.net/

IRCOMM: Serial and Parallel Port Emulation over IR -article,
IrDA FTP Site. (ircomm10.doc)
ftp.irda.org

Writing Great IrDA Applications.
Microsoft Website

Programming Microsoft Windows CE by Douglas Boling.
Microsoft Press